P. Findeisen, P. Gburzynski

G A S S

MERA-400 ASSEMBLY LANGUAGE

R E F E R E N C E M A N U A L

VERSION 6.0

PART TWO

IIUW, WARSZAWA, 1981

REVISED IN NOVEMBER 1987

The list of contents

i. System configuration 3

ii. Memory organization 4

iii. The processor 5

iv. Interrupts 7

v. Machine instruction execution 10

vi. Instructions formats 12

vii. The instructions list 14

vii.1 System instructions 16
vii.2 Miscellaneous instructions 22
vii.3 Load & store instructions 27
vii.4 Fixed point arithmetic 33
vii.5 Logical instructions 37
vii.6 Shifts 40
vii.8 Comparing instructions 47
vii.9 Byte instructions 48
vii.9 Conditional instructions 51
vii.10 Branching instructions 56
vii.21 Floating point arithmetic 61

Appendix A 63

Appendix B 64

Appendix C 68

Special thanks to P.T.Raszkowski who has typed in a preliminary version of the instruction list.

i. System configuration

The heart of the Mera-400 minicomputer is the interface bus. There are four basic modules that can be connected to the bus:

- the memory physical block
- the processor
- the direct memory access channel
- the byte channel

The memory and the processor are described in the next sections.

The direct memory access channel is provided for fast input/output operations, usually for using mass storage media like disc or magnetic tape. The speed of channel transmission cannot exceed 500000 words/second. The principle of processor-channel cooperation is that the processor initializes the transmission (or another operation) and is interrupted when the operation is completed. Accessing memory is performed by the channel automatically. The channel is able to perform only one transmission at a time.

The byte channel is designed for slow devices such as tty, paper tape reader, floppy disc etc. It provides the possibility of data transmission in words or in bytes, up to 15000 per second. The input/output request can sometimes be answered immediately, within a single instruction cycle. In that case the channel doesn't interrupt the processor. For details see input/output instruction description and the peripheral interface tables.

A channel of any type can serve for up to 8 control units. A control unit is the smallest unit addressable by i/o operations.

The total number of channels must not be greater than 16, and only two processors are permitted in the configuration. This limitation is caused, however, by technical details, and one can assume, without loss of generality, that the number of processors is arbitrary.

ii. Memory organization

The memory is organized into physical blocks. Blocks are divided into 4096-word frames. Each block can contain up to 8 frames. The word consists of 16 bits numbered from left to right. The leftmost bit is bit zero.

There are two 4-bit registers associated with every frame: space index register and page number register. When a word is to be fetched or to be stored two values must be provided by the interface bus: a 16-bit address and a 4-bit space index. The space index and the four upper address bits are compared associatively against the contents of all of the space index and page number registers, and the frame that provides equality is selected. The lower twelve bits of the address are used as offset in the selected frame.

It follows from the above that there should not exist a pair of frames with the same contents in their associated registers. Initially, all these 4-bit registers contain B'1111', however.

Pages number zero and one of the address space of index zero are connected directly to the processor and are accessed without the help of the matching algorithm. Thus the word addresses 0..8191 of the zero address space are always valid and, what is of great importance, determine distinct objects for distinct processors.

The zero address space is provided for the use of the operating system and will be called system address space.

Thus it is possible to form one system and fifteen user address spaces. Each address space can include up to 65536 words. The amount of memory connected to the bus in the single configuration is limited, however, to 16 physical blocks.

See also descriptions of the **reset** and **o.rq** instructions.

There exist also non-standard memory blocks, which don't exactly provide the facilities described above. Please consult your memory technical manual to check whether your core fits to the standard category.

iii. The processor

Each processor is equipped with the following registers:

IC Instruction counter (16 bits). During the machine operation execution IC points to the next operation to be performed.

PM Processor mask (10 bits). This register serves as a mask for the hierarchical interrupt system.

M Mode bit. When this bit is set, the minicomputer acts in user mode. Otherwise the computer is in the system mode. In user mode the so called 'privileged' instructions are illegal.

A Access bit. This bit can be used in two-processor configuration only. If the bit is set all references to the bus (performed by the given processor) have the processor number reversed (e.g. The processor number 1 simulates the behaviour of the processor number 0). The effects of this are serious - in particular when the contents of IC is less than 8192 the instructions are fetched from the other processor address space.
The concept of the **A** bit seems awkward and the authors have never used this gadget.

SI Space index (4 bits). This register usually contains the address space index for the current user program.

PS Processor status (16 bits). The processor status is formed by **PM**, **M**, **A** and **SI** register.

IV Interrupt vector (32 bits). The particular bits of interrupt vector are set by corresponding events as described in section iv.

r1-r7 Universal 16-bit registers. They can be used for arithmetic and logical operations and for indexing.

r0 (called also **PF**) Program flags. This register can be referenced as the first argument of operations, but never as the second one nor for indexing. Its nine leftmost bits are individually set and tested by many instructions. Theese bits are:

Z - r0<0> Zero flag. Set if the result of the last arithmetic or logical instruction is zero.

N - r0<1> Negative flag. Set if the result of the last arithmetic instruction is negative.

V - r0<2> Overflow flag. Set when an arithmetic instruction produces an overflowing result. This bit is never cleared by an arithmetic instruction.

C - r0<3> Carry flag. Set when a carry occurs during an arithhmetic operation.

L - r0<4> Less than flag.
E - r0<5> Equal to flag.
G - r0<6> Greater than flag. Set by the comparing instructions.

H - r0<7> Shift flag. This bit is used by shift operations.

X - r0<8> Auxiliary flag. Used by shift operations. Can also be directly set by the user.

If the mode bit is set all instructions that specify r0 as destination of the result (except the **mrpf** instruction) do not affect the left byte of r0. However the lower eight bits of the result will be stored in the right byte. Thus the right byte of r0 is sometimes called **SB** (scratch byte) and the left byte - **AF** (arithmetic flags).

S Skip indicator. If this bit is set the next instruction will not be executed (i.e. it will be skipped) and the skip indicator will be cleared.

AM Argument modifier (16 bits). This is a virtual register that holds the second argument value of the **setam** instruction. The value is used for the second argument modification of the next instruction as described in the instruction list (see **setam** description).

PR Panel register (16 bits). The panel register consists of the 16 switches, which can be set and freely altered by the user. They cannot be affected in any way by a computer program.

iv. Interrupts

As was already mentioned, each processor is equipped with a 32-bit interrupt vector. The vector can be divided logically into two subregisters:

PI	=	IV<0..11,28..31>	processor interrupts
CI	=	IV<12..27>	channel interrupts

This is not the right place for explaning the role played by the interrupt system in a modern computer, so let us lunge straight into the details at hand.

Here are the events that cause the particular bits of the interrupt vector to be set:

0 - power alarm of the central processor unit. This position of the interrupt vector cannot be masked, i.e. it always triggers an interrupt. After the event occurs, the power supply should be able to work for approx. 1 ms.

1 - memory parity error. Each 16-bit word is in fact represented as a 17-bit string; the auxilary bit is a parity bit, computed and stored automatically. When a word with an invalid parity bit is being fetched from memory the interrupt bit is set. Moreover if the parity error occurs in the fetched instruction when the mode bit is clear the computer stops and the alarm indicator on the panel is beeing turned on.

2 - memory missing. If the processor attempts to reference a location that is absent in the current address space this bit is set. As in the previous case the computer is stopped when the mode bit is clear and an instruction fetch is being attempted.
The reader should be warned here that information supplied by this interrupt seems insufficient to organize the virtual address space.

3 - high signal from another processor. This event can be caused by another processor executing the instruction **if.hsi**.

4 - interface power alarm. When the power supply of any computer's module (other than the processor unit) connected directly to the interface bus is turned off or on the bit is set.

5 - clock interrupt. Each processor is equipped with a clock. This clock can be turned off by the corresponding switch on the computer panel. When the clock is running it sets the interrupt bit every 10 ms. This time can be changed by the hardware maintenance service, however.

6 - illegal instruction. The bit is set when the processor is unable to decode the fetched instruction, or the instruction is reserved for the operating system and the mode bit is set. This bit is never set if the skip indicator is on.

7 - fixed-point division overflow. See the **div** instruction for details.

8 - floating-point underflow. See the floating point instructions for details.

9 - floating point overflow. See the floating point instructions for details.

10 - division by zero or unnormalized floating-point argument. See the floating point instructions for details.

11 - not used (reserved for future extensions).

28 - operator intervention. There is a special switch on the computer panel marked **OPRQ**. When this switch is pressed the 28th bit of IV is set.

29 - low signal from another processor. Similarly as in the case of bit no. 3 another processor can set this bit by the instruction **if.lsi**.

30 - high program interrupt. The bit is set by the instruction **rhpi** or **rbpi**.

31 - low program interrupt. The bit is set by the instruction **rlpi** or **rbpi**.

Positions **12 - 27** of the interrupt vector are reserved for **channel interrupts**. Position no. 12 corresponds to channel no. 0, position no. 13 to channel no. 1 and so on.

The interrupts are divided into 11 **levels**. It is convenient to count levels from -1 to 9:

level	interrupts
-1	0
0	1
1	2
2	3
3	4
4	5-11
5	12-13
6	14-15
7	16-21
8	22-27
9	28-31

The interrupt of level -1 cannot be masked. In the remaining cases the level of interrupt determines the associated bit of the processor mask which controls the interrupt. When the interrupt bit and its corresponding mask bit are both set the interrupt is triggered. However in some cases these conditions are insufficient. An interrupt cannot be triggered if the skip indicator is set or if the last instruction executed was **setam** (see its detailed description).

The **hardware interrupt service** proceeds as follows:

The contents of cell 97 of the system address space is fetched and is used as a pointer to the first free position on the stack. The following

4 values are pushed onto the stack: instruction counter, program flags, processor status and interrupt specification. The interrupt specification is significant only in case of channel interrupts, otherwise it is zero. The stack pointer (incremented by 4) is then stored back in location 97. The stack is located in the system address space.

The interrupt vector bit that caused the interrupt is cleared. The corresponding mask bit and all mask bits of indices greater than the level of interrupt in question are also cleared. Finally r0 and the mode bit are cleared.

The new value of the instruction counter is determined by the contents of the location computed as the sum of the interrupt index and 64.

The procedure described above is indivisible, i.e. no other interrupt is triggered in the meantime. If two different interrupts can be triggered the lower index interrupt always takes the precedence.

The channel interrupt specification is formed from two values: the interrupt specification code (bits 0-7) and the control unit number (bits 8 - 10). The bits 11 - 15 of the interrupt specification are insignificant. The interrupt specification code depends on the kind of control unit used. The channel always interrupts the processor which executed the i/o request resulting in the interrupt.

v. Machine instruction execution

The processor, as everyone knows, performs machine instructions in natural order, with the exception of branch statements and interrupts. This mechanism is implemented in the following steps:

1 - In the first step the S, AM, IV and PM registers are tested to determine whether the hardware interrupt routine is to be performed. If not, the processor continues with step 2.

2 - The first word of the instruction is fetched from the memory location addressed by the instruction counter, and IC is incremented by one. If the mode bit is cleared, the system address space is used; otherwise the space determined by the contents of SI is used. If this step fails, bit 1 or 2 of IV is set (see interrupts section for details), and the processor returns back to the first step. Note that the IC incrementation cannot fail.

3 - The instruction is decoded. If the instruction opcode is invalid or the mode bit is set and the instruction is a system-reserved one, it is considered invalid. In such a case if the S indicator is cleared, then bit 6 of IV is set. The next step executed will be step one (sic!).
When the instruction is valid, the hardware checks whether its length is one or two words. In the latter case the memory location determined by IC is fetched and IC is incremented. The address space identification and the reaction to memory failure is identical as in step 2.

4 - If the S indicator is cleared the second argument is evaluated and the instruction is executed, otherwise S is cleared. Notice that instruction execution can change IC, set S, AM or an interrupt bit. After this step is performed, the processor executes step 1.

If the mode bit is set the address space determined by the contents of SI (the indicated address space for short) is used for evaluation of the second argument. If the mode bit is cleared the system address space is used. However, there are some exceptions from this rule. Some instructions (always with DI format arguments) specify the indicated address space independently of the mode bit. In such cases, however, if double indirection is specified, its first step, i.e. fetching the address of final argument, is performed according to the first rule.

Example

Let us assume that word 120 of the system address space contains the value 7104. Then if the mode bit is cleared the execution of the instruction:

```
store. r1, ((120))
```

stores the register r1 in word 7104 of the address space determined by the actual contents of SI.

For a more detailed description of processor activity one can consult Mera-400 technical manuals.

vi. Instructions formats

It follows from part one of this manual that the following formats of an instruction's second argument are accepted:

SI - single indirect
DI - double indirect
SR - short relative
SN - short numeric
BA - byte argument

The syntactical form of all those arguments has already been described. Now it is time to say a few words about the internal representation of machine instructions.

A machine instruction always starts on word boundary and its length may be one or two words.

All SI and DI argument instructions have identical representation:

bits 0 - 5 The opcode

bit 6 Indirect modification bit (additional indirection when set)

bits 7 - 9 First argument register designator or the opcode extension

bits 10 - 12 Index register designator, or zero if indexing is not used

bits 13 - 15 Register designator of the second argument or zero if an expression of any form has been used. In this case the instruction is two words long and the expression is assembled into the second word.

Let us take the **sub** instruction as an example. Its opcode is O'42' (or in binary, B'100010').

Thus the following instructions will be assembled as shown:

```
sub   r4, r6            ; 100010 0 100 000 110
sub   r7, (r4-1)        ; 100010 1 111 100 000 FFFF
sub   r4, 3             ; 100010 0 100 000 000 0003
sub   r0, (r2+r4)       ; 100010 1 000 010 100
```

(the second word of the instruction is written in hexadecimal form.)

The SR and SN argument instructions also have identical representation:

bits 0 - 5 The opcode

bit 6 Sign of the second argument's value (0 for +, 1 for -)

bits 7 - 9 First argument register designator or the opcode extension

bits 10 - 15 Second argument absolute value

As an example let us take the instruction **b.cs**. Its opcode reads O'70' (binary '111000') and the extension is B'111'. The assembly will look as follows:

```
        b.cs       .            ; 111000 0 111 000000
        b.cs       -2           ; 111000 1 111 000010
        b.cs       .(5)         ; 111000 0 111 000101
```

The primary value of the SN or SR argument is extended to a signed 16-bit number before any other computations take place.

The last argument format is BA. It is represented as follows:

bits 0 - 5 The opcode

bits 6 - 7 The opcode extension

bits 8 - 15 The argument value

Instructions without a second argument, but with a first argument have the following format:

bits 0 - 5 The opcode

bits 6, 10 - 15 The opcode extension

bits 7 - 9 The register designator

Finally, instructions without arguments have the format:

bits 0 - 5 The opcode

bits 6 - 15 The opcode extension

In the last three cases no examples were shown. The reader is expected to learn more about internal instruction representation from the instruction list, where the place reserved for the argument is marked by dashes.

In some cases particular bits of the opcode extension are insignificant. Such bits are also marked by dashes in the instruction list.

vii. The instructions list

The Mera-400 instructions list is not very different from that of other minicomputers. However, some instructions seem useless - they are only slight variations of other instructions. On the other hand one can look for many f other instructions. On the other hand one can look for many useful instructions unsuccessfully.

In general, addressability in Mera-400 is not as good as one can expect from a minicomputer. There are no auto-increment or auto-decrement addressing modes. It is possible, however, to use index registers and indirect addressing. The space of logical addresses is limited to 64K words.

The following pages contain the full Gass instruction list divided into several groups. The instruction descriptions are announced in the left margin by symbolic opcodes with the arguments descriptors in our usual notation. The mnemonic is then explained and the bit pattern of the internal machine representation is given (the opcodes appear as octal numbers). The dashes stand for arguments or for insignificant bits. Some Gass instructions are variants of other instructions (with one or two arguments fixed). These cases are marked by an '=' character followed by the original instruction.

The descriptions also contain the symbolic notation of instruction semantics, and more detailed explanation if necessary.

In a description of instruction semantics the register used as the first argument is denoted by **R**. The brackets < > denote bit position bounds in the specified item. The single letter **Q** stands for the first word of the second argument (or the primary value in case of a direct argument); in special cases, when the second argument is defined by a word (or words) in the indicated address space, **Q'** is used. The parentheses ( ) are used for indirect addressing in the sense of Gass assembly language.

To understand the symbolic notation for instruction semantics one must remember that Q doesn't denote a constant that can be derived from the instruction. In the case of the first or the second indirect format of the second argument it always represents a memory word, which should be treated as a variable in high-level programming languages.

Example

Let us consider three sample second arguments that differ by the level of indirection:

1. r3-12

Q represents here a value which is computed as the difference between the contents of r3 and 12.

2. Dobo(1)

Q represents here a memory word (a variable). The address of the word is computed as the sum of the value of the address symbol Dobo and one.

3. (Jolly(r7-6))

Here Q stands for the memory word, whose address is fetched from the memory word addressed by the sum of the value of the address symbol Jolly, the contents of r7 and the value -6.

The expression **Q(N)** denotes a memory word located N words forward from the word Q.

vii.1 System instructions

All system instructions except input/output requests are invalid in user mode. If the user tries to execute an i/o operation directly, it usually doesn't give any desired effect, but no 'invalid instruction' interrupt appears. However, the hardware is open for construction of a peripheral device that can be controlled by the user program.

The system instructions are designed to handle the processor status register, the interrupt vector and the i/o devices.

Some of the system instructions seem awkward. For example, it is impossible to set space index register to any requested value without involving an auxilary cell in core.

.....

o.rq R,SI

```
   >>> output request

       35 - --- ------

       if Q<15>=0
       then request output:
            ( channel       = Q<11..14>
              unit          = Q<8..10>
              operation     = Q<0..7>
              register      = R
            )
       else address memory:
            ( physical memory block        = Q<11..14>
              physical frame number        = Q<8..10>
              new address space index      = R<12..15>
              new logical page number      = R<0..3>
            )
       case of
          no response  : IC := (IC)
          engaged      : IC := (IC+1)
          ok           : IC := (IC+2)
          parity error : IC := (IC+3)
       end
```

There are two kinds of output request operands. One is used for creating address space from available memory frames, the second provides co-operation with peripherals.

Let us focus first on the memory allocation details. As was already mentioned, two registers are associated with each memory frame. If we want to update these registers, we must have some means for referencing the frame explicitly. This feasibility is provided only by the **o.rq** instruction. If the value of the second argument is odd it is assumed that it describes a frame address as follows:

```
bits 11 - 14 of the value        - physical block number
bits  8 - 10 of the value        - frame number in block
```

The first argument contains the new logical space index and page number to be saved in associated registers. The logical space index is placed in bits 12 - 15 and the page number in bits 0 - 3 of the

specified register.
The frames which contain pages zero and one of the system address space cannot be re-addressed.

Example

Assume that one wants to delete frame 5 of physical block 1 from an address space. Then the following sequence does the job:

```
lds     r1, -1        ; R1<0..3> and R1<12..15> are all-ones
o.rq    r1, X'A3'
```

If the value of second argument is even it is used to reference the particular control unit:

```
bits  0 -  7        -  internal function code
bits  8 - 10        -  control unit number
bits 11 - 14        -  channel number
```

The internal function codes and the details of register usage depend on the kind of control unit used and can be obtained from special tables.

There are four possible reactions of the unit (device or memory frame) referenced in an input/output instruction:

- no response — No reaction at all. This may happen if the address of the unit is invalid, the unit is out of order, its power supply is turned off or the instruction is executed in user mode.

- engaged — The unit is not ready to execute the desired action. It is not true that if the unit is not ready a reference to it has no effect at all. Sometimes the unit starts to act in a way that will allow to execute the requested action in the future (for an example see the description of paper tape reader control unit).

- ok — The unit has executed the desired function successfully.

- error — The unit has executed the desired function but parity error has occured during data transmission.

One of the peculiarities of the input/output instruction is the way it affects the instruction counter. Depending on the reaction of the referenced unit, IC is changed as follows:

```
no response          IC := (IC)
engaged              IC := (IC+1)
ok                   IC := (IC+2)
error                IC := (IC+3)
```

In practice it means that the programmer has to put after each input/output statement four addresses to which he wants the processor control to branch on each condition.

.....

i.rq R,SI
>>> input request

```
36 - --- ------

if Q<15> = 0
then request input:
     ( channel       = Q<11..14>
       unit          = Q<8..10>
       operation     = Q<0..7>
       register      = R
     );
case of
   no response  : IC := (IC)
   engaged      : IC := (IC+1)
   ok           : IC := (IC+2)
   parity error : IC := (IC+3)
end
```

The input request behaves similarly to the **o.rq** instruction, except that it doesn't have the memory reference variant.

.....

halt
>>> stop the computer

```
73 - 000 ------
```

After executing a halt instruction the processor stops and doesn't fetch any more instructions for execution until an interrupt is triggered. Then a default hardware action takes place and the processor resumes running.

.....

exit
>>> exit interrupt routine

```
73 - 100 ------

IC := ((97)-4)
r0 := ((97)-3)
PS := ((97)-2)
(97) := (97) - 4
```

The instruction restores IC, r0 and PS from the interrupt stack. It is usually executed as the exit statement of an interrupt routine.

.....

swch. DI
>>> switch to another program

```
77 - 100 ------
```

```
          IC := Q'
          r0 := Q'(1)
          PS := Q'(2)
```

Ic, r0 and PS are loaded simultaneousely with the second argument's value. The indicated address space is used for the second argument evaluation.

.....

rbpi

>>> request both program interrupts

```
          73 - 010 ----11

          IV<30..31> := B'11'
```

The two rightmost bits of IV are set.

.....

rhpi

>>> request high program interrupt

```
          73 - 010 ----10

          IV<30> := 1
```

The thirtieth bit of IV is set.

.....

rlpi

>>> request low program interrupt

```
          73 - 010 ----01

          IV<31> := 1
```

The thirty first bit of IV is set.

.....

cbpi

>>> clear both program interrupts

```
          73 - 010 ----00

          IV<30..31> := 0
```

The two rightmost bits of IV are cleared.

.....

ldsi DI

>>> load space index

```
          77 - 000 ------
```

```
          PS<10..15> := Q<10..15>
```

Bits 10..15 of the second argument's value are stored in the bits 10..15 of PS (i.e. M, A and SI).

.....

ldpm DI

>>> load processor mask

```
          77 - 001 ------

          PS<0..9> := Q<0..9>
```

Bits 0..9 of the second argument's value are stored in the mask register.

.....

stpi DI

>>> store processor interrupts

```
          77 - 010 ------

          Q := IV<0..11,28..31>
```

Bits 0..11 and 28..31 of the interrupt vector are stored in the second argument.

.....

ldpi DI

>>> load processor interrupts

```
          77 - 011 ------

          IV<0..11,28..31> := Q
```

The second argument's value is stored in bits 0..11 and 28..31 of IV.

.....

if.hsi

>>> if high interrupt sent and ignored

```
          73 0 011 ------

          IV<3> (* of another processor *) := 1
          if success then S := 1
```

The instruction sends an interrupt signal to all the other processors. This signal causes bit 3 of IV to be set. If the interrupt signal is accepted by a processor it responds with the answer 'ok'. If at least one positive answer is received the skip indicator is set.

.....

if.lsi

```
   >>> if low interrupt sent and ignored

       73 1 011 ------

       IV<29> (* of another processor *) := 1
       if success then S := 1
```

The instruction acts in the way described above except that its interrupt signal sets the 29th bit of IV rather than the 3rd.

.....

reset

```
   >>> reset the configuration

       73 - 001 ------

       clear channels,
       reset memory allocation,
  (* and for all processors: *)
       r0 := 0
       PS := 0
       IV := 0
```

This instruction "excludes" all pages (with the exception of the pages in processors private frames) from address spaces. The frames will contain all-ones in their associative registers.

All the device control units are reset. This is similar (but more powerful) to the sequence of output statements issueing the request 'clear' to all the control units available in the actual configuration. R0, PS and IV of all processors are also cleared.

.....

vii.2 Miscellaneous instructions

In most computers there is a lot of instructions that cannot be attached to any reasonable instruction group. The instructions described here usually do some simple operations on the universal and special registers of the computer.

The two non-trivial instructions in this group are **ptr** and **setam**.

ptr BA

```
>>> program trap

    71 0 1-- ------

    ((97)) := IC
    ((97)+1) := r0
    ((97)+2) := PS
    ((97)+3) := Q
    IC := (96)
    r0 := 0
    PS<9..10> := 0
    (97) := (97) + 4
```

The ptr instruction is designed for communication between a user program and the operating system. The execution of this instruction is similar to the hardware reaction on an interrupt, except that no mask is involved here, and the new instruction counter value is taken from the 96th word of the system address space. The value of the second argument is used instead of an interrupt specification.

.....

clv

```
>>> clear V

    = b.vs     .
```

The V indicator of r0 is cleared.

.....

cllb R

```
>>> clear left byte

    72 0 --- 000001

    R<0..7> := 0
```

The left byte of the specified register is cleared.

.....

clrb R

```
>>> clear right byte

    72 1 --- 000001
```

```
          R<8..15> := 0
```

The right byte of the specified register is cleared.

.....

swab R

```
   >>> swap bytes

          = src.8  R
```

The left and the right bytes of the register are interchanged.

.....

clr R

```
   >>> clear register

          = lds     R,0
```

The specified register is cleared.

.....

clrm DI

```
   >>> clear memory word

          77 - 110 ------

          Q := 0
```

A single word denoted by the second argument is cleared.

.....

incm DI

```
   >>> increment memory word

          = if.anz    DI
```

The contents of specified memory word is incremented. Note that if result of the incrementation is zero, next instruction is skipped. The instruction 'incm' is identical as the instruction 'if.anz'.

.....

mld R,SI

```
   >>> mask & load

          22 - --- ------

          R :=  R .and. .not.r7   .ior.  Q.and.r7
```

The bits of r7 that have the value of one force the corresponding bits of the second argument's value to be copied into R. The other bits of R remain unaltered.

.....

push R,SI
>>> push value onto stack

23 - --- ------

(R) := Q
R := R + 1

The second argument's value is stored in the memory location defined by the address contained in R and the register is incremented.

.....

msx R
>>> move sign bit to X

72 0 --- 000010

X := R<0>

The X indicator is set according to the sign bit of the short integer contained in the register.

.....

mpx R
>>> move parity bit to X

72 1 --- 000010

X := R<15>

The rightmost bit of the register is copied to the X indicator.

.....

mrpf R
>>> move the register to program flags

72 1 --- 001001

r0 := R

The contents of the specified register is stored in r0. This is the only instruction that can make global change of AF in user mode.

.....

mpfr R
>>> move program flags to the register

72 0 --- 001001

R := r0

R0 is stored in the specified register.

.....

nop

>>> no operation

= mrpf r0

The nop instruction doesn't change the environment but increments the instruction counter.

.....

micr R

>>> move instruction counter to the register

72 0 --- 000000

R := IC

The value of IC is stored in the register. Note that IC incrementation takes place before copying.

.....

mprr R

>>> move panel register to the register

72 1 --- 000000

R := pr

The contents of the panel register (switches) is copied to the register.

.....

setam SI

>>> set argument modifier

77 - 101 ------

AM := Q

The value of the second argument is saved as argument modifier. This value is then added to the primary value of the next instruction's second argument and the sum is used instead of the primary value. The argument modifier has influence on the next instruction argument only; after modification it is cleared. If the next instruction doesn't have a second argument, the argument modifier is lost. The modification of another setam instruction is possible, however, only sequences of up to three setam instructions are permitted. The fourth setam instruction is considered invalid and sets the 6th bit of interrupt vector. If the argument modifier is set no interrupt is accepted.

.....

vii.3 Load & store instructions

There are pairs of load and store instructions that differ in details only. Some of these instructions could probably be deleted from the list without any serious consequences.

l.rg R,SR
>>> load register

66 - --- ------

R := (IC + Q primary value)

The memory location described by the sum of IC and the primary value of the second argument is stored in the register.

.....

s.rg R,SR
>>> store register

67 - --- ------

(Q primary value + IC) := R

The contents of the register is stored in the memory word described by the sum of IC and the primary value of the second argument.

.....

lds R,SN
>>> load short

65 - --- ------

R := Q

The specified constant is placed in the register.

.....

set R,SI
>>> set register

20 - --- ------

R := Q

The second argument's value is stored in the register.

.....

store R,DI
>>> store register

24 - --- ------

```
          Q := R
```

The value of the register is stored in the second argument.

.....

load. R,DI

```
   >>> load register

       21 - --- ------

       R := Q'
```

The word described by the second argument is stored in the register. The indicated address space is used for fetching the second argument.

.....

store. R,DI

```
   >>> store register

       25 - --- ------

       Q' := R
```

The register is stored in the second argument. The second argument is located in the indicated address space.

.....

ldl DI

```
   >>> load long

       75 - 000 ------

       r1_r2 := Q(0)_Q(1)
```

The long integer defined by the second argument is placed in registers r1 and r2.

.....

strl di

```
   >>> store long

       76 - 000 ------

       Q(0)_Q(1) := r1_r2
```

The long integer located in r1 and r2 is stored in the second argument.

.....

ldl. DI

```
   >>> load long

       75 - 100 ------
```

```
          r1_r2 := Q'(0)_Q'(1)
```

The long integer defined in the indicated address space by the second argument is loaded to registers r1 and r2.

.....

strl. DI

```
   >>> store long

          76 - 100 ------

          Q'(0)_Q'(1) := r1_r2
```

The long integer from r1 and r2 is stored in the second argument. The indicated address space is used.

.....

ldf DI

```
   >>> load floating

          75 - 001 ------

          r1_r2_r3 := Q(0)_Q(1)_Q(2)
```

The floating-point value is moved from the second argument to the registers r1, r2 and r3.

.....

strf DI

```
   >>> store floating

          76 - 001 ------

          Q(0)_Q(1)_Q(2) := r1_r2_r3
```

The floating-point value from the registers r1, r2 and r3 is stored in the second argument.

.....

ldf. DI

```
   >>> load floating

          75 - 101 ------

          r1_r2_r3 := Q'(0)_Q'(1)_Q'(2)
```

The floating-point value is fetched from the second argument. The indicated address space is used.

.....

strf. DI
>>> store floating

76 - 101 ------

Q'(0)_Q'(1)_Q'(2) := r1_r2_r3

The floating-point value from the registers r1, r2 and r3 is stored in the second argument. The indicated address space is used.

.....

lda DI
>>> load all

75 - 010 ------

r1_r2_r3_r4_r5_r6_r7 := Q(0)_Q(1)_Q(2)_Q(3)_Q(4)_Q(5)_Q(6)

The seven words of second argument are loaded into the universal registers.

.....

stra DI
>>> store all

76 - 010 ------

Q(0)_Q(1)_Q(2)_Q(3)_Q(4)_Q(5)_Q(6) := r1_r2_r3_r4_r5_r6_r7

The universal registers are stored in the seven consecutive words of second argument.

.....

lda. DI
>>> load all

75 - 110 ------

r1_r2_r3_r4_r5_r6_r7_ :=
Q'(0)_Q'(1)_Q'(2)_Q'(3)_Q'(4)_Q'(5)_Q'(6)

The seven words of second argument are placed in the registers. The indicated address space is used.

.....

stra. DI
>>> store all

76 - 110 ------

Q'(0)_Q'(1)_Q'(2)_Q'(3)_Q'(4)_Q'(5)_Q'(6) :=
r1_r2_r3_r4_r5_r6_r7

The universal registers are stored in the second argument. The indicated address space is used.

.....

ldh DI
>>> load high registers

75 - 011 ------

r5_r6_r7 := Q(0)_Q(1)_Q(2)

The three words of second argument are placed in r5, r6 and r7.

.....

strh DI
>>> store high

76 - 011 ------

Q(0)_Q(1)_Q(2) := r5_r6_r7

The registers r5, r6 and r7 are stored in the second argument.

.....

ldh. DI
>>> load high

75 - 111 ------

r5_r6_r7 := Q'(0)_Q'(1)_Q'(2)

The second argument is loaded to the registers r5, r6 and r7. The indicated address space is used.

.....

strh. DI
>>> store high

76 - 111 ------

Q'(0)_Q'(1)_Q'(2) := r5_r6_r7

The registers r5, r6 and r7 are stored in the second argument. The indicated address space is used.

.....

vii.4 Fixed-point arithmetic

There are two kinds of integers handled by the hardware: short (one word) and long (two words). Numbers are represented in binary two-complement notation.

The ranges for integers are:

```
           -32768..32767            for short integers
  and
      -2147483648..2147483647       for long integers
```

A one-word integer can represent an address, the sign bit has the weight of 2**15 then, and therefore the address range is 0..65535.

Available operations are additions and subtractions, both in the short and the long version, multiplication and division. Multiplication takes two short arguments and results in a long quantity. Division requires a long dividend and a short divisor. The result consists of two short values: an integer quotient and a remainder.

The short addition and subtraction operations can be also used for address computation.

The arithmetic operations set the suitable flags according to their results. the flag values become 0 or 1, which indicates false and true respectively. The V indicator can be set to 1 only by these operations, i.e. V can be tested at the end of an arithmetic instructions sequence and will show if any step of the calculation has failed (but it should be cleared before computation).

ngn R

>>> negate numeric value

```
72 0 --- 000011

R := - R
flags: Z,N,V,C
```

The sign of the short integer value in the register is reversed.

.....

add R,SI

>>> add

```
40 - --- ------

R := R + Q
flags: Z,N,V,C
```

The second argument's value is added to the register.

.....

adds R,SN

>>> add short

```
60 - --- ------
```

```
          R := R + Q
          flags: Z,N,V,C
```

The second argument's value is added to the register.

.....

inc R

```
   >>> increment

          = adds   R,1
```

The specified register is incremented by one.

.....

sub R,SI

```
   >>> subtract

          42 - --- ------

          R := R - Q
          flags: Z,N,V,C
```

The second argument is subtracted from the register.

.....

subs R,SN

```
   >>> subtract short

          = adds   R,-SN
```

The second argument is subtracted from the register.

.....

dec R

```
   >>> decrement

          = adds   R,-1
```

The specified register is decremented by one.

.....

addl DI

```
   >>> add long

          37 - 000 ------

          r1_r2 := r1_r2 + Q(0)_Q(1)
          flags: Z,N,V,C
```

The long integer value of the second argument is added to the long integer in registers r1 and r2.

.....

subl DI
>>> subtract long

37 - 001 ------

r1_r2 := r1_r2 - Q(0)_Q(1)
flags: Z,N,V,C

The long integer value of the second argument is subtracted from registers r1 and r2.

.....

mul DI
>>> multiply

37 - 010 ------

r1_r2 := r2 * Q
flags: Z,N

The short integer in register r2 is multiplied by a short integer stored in the second argument. The result appears as a long integer in registers r1 and r2.

.....

div DI
>>> divide

37 - 011 ------

r2 r1_r2 / Q
:=
r1 r1_r2 \ Q
flags: Z,N

The long integer in registers r1 and r2 is divided into the short integer stored in the second argument. A short integer result is placed in r2 and the remainder in register r1. The remainder always has the sign of the divisor. If the quotient exceeds the range for short integers the 7th bit of the interrupt vector is set.

.....

adc R,SI
>>> add with carry

41 - --- ------

R := R + Q + C
flags: Z,N,V,C

The sum of the second argument's value and of the carry bit (0 or 1) is added to the register.

.....

ngc R

>>> negate with carry

72 1 --- 000011

R := C - 1 - R
flags: Z,N,V,C

The sign of the short integer stored in the register is reversed. If the carry bit was cleared the result is decremented by one.

.....

vii.5 Logical instructions

The logical instructions provide a set of tools for manipulating 16-bit patterns treated as vectors of boolean values.

rvb R
>>> reverse bits

```
72 0 --- 001000

R := .not. R
flags: Z
```

Every bit of the specified register is reversed.

.....

and R,SI
>>> logical and

```
46 - --- ------

R := R .and. Q
flags: Z
```

The logical bit by bit product of the contents of the register and the second argument's value is computed and stored in the register.

.....

andm. R,DI
>>> logical and in memory

```
47 - --- ------

Q' := Q' .and. R
flags: Z
```

The bit-by-bit logical product of the register and the second argument value is computed and stored in the second argument. The indicated address space is used.

.....

ior R,SI
>>> logical inclusive or

```
44 - --- ------

R := R .ior. Q
flags: Z
```

The bit by bit logical sum is computed and saved in the register.

.....

iorm. R,DI
>>> logical inclusive or in memory

45 - --- ------

Q' := Q' .ior. R
flags: Z

The bit by bit logical sum of the second argument's value and the register is computed and stored in the second argument. The indicated address space is used.

.....

xor R,SI
>>> logical exclusive or

52 - --- ------

R := R .xor. Q
flags: Z

The bit by bit lo R := R .xor. Q
flags: Z

The bit by bit logical exclusive or of the register and the second argument value is computed and stored in the register.

.....

xorm. R,DI
>>> logical exclusive or in memory

53 - --- ------

Q' := Q' .xor. R
flags: Z

The bit by bit logical exclusive or of the second argument's value and the register is computed and the result is stored in the second argument. The indicated address space is used.

.....

erb R,SI
>>> erase bits

50 - --- ------

R := R .and. .not.Q
flags: Z

The bit by bit logical difference of the register and the second argument value is computed and stored in the register.

.....

erbm. R,DI
>>> erase bits in memory

51 - --- ------

```
Q' := Q' .and. .not.R
flags: Z
```

The bit by bit logical difference of the second argument's value and the register is computed and the result is stored in the second argument. The indicated address space is used.

.....

vii.6 Shifts

The set of shifts available in Mera-400 doesn't contain any instruction with variable displacement, and the only instructions that can shift a register more than one position are circular shifts. Unfortunately the displacement value for these instructions is contained in the operation code extension and that makes the circular shift list as queer as follows:

src.1 R
>>> shift right circular by 1 position

72 0 --- 010001

R := R<15>_R<0..14>

.....

slc.15 R
>>> shift left circular by 15 positions

= src.1 R

.....

src.2 R
>>> shift right circular by 2 positions

72 0 --- 010010

R := R<14..15>_R<0..13>

.....

slc.14 R
>>> shift left circular by 14 positions

= src.2 R

.....

src.3 R
>>> shift right circular by 3 positions

72 0 --- 010011

R := R<13..15>_R<0..12>

.....

slc.13 R
>>> shift left circular by 13 positions

= src.3 R

.....

src.4 R
   >>> shift right circular by 4 positions

       72 0 --- 010100

       R := R<12..15>_R<0..11>

.....

slc.12 R
   >>> shift left circular by 12 positions

       = src.4 R

.....

src.5 R
   >>> shift right circular by 5 positions

       72 0 --- 010101 ft right circular by 5 positions

       72 0 --- 010101

       R := R<11..15>_R<0..10>

.....

slc.11 R
   >>> shift left circular by 11 positions

       = src.5 R

.....

src.6 R
   >>> shift right circular by 6 positions

       72 0 --- 010110

       R := R<10..15>_R<0..9>

.....

slc.10 R
   >>> shift left circular by 10 positions

       = src.6 R

.....

src.7 R
   >>> shift right circular by 7 positions

       72 0 --- 010111

       R := R<9..15.>_R<0..8>

.....

slc.9 R
   >>> shift left circular by 9 positions

```
          = src.7  R
```

.....

src.8 R

```
   >>> shift right circular by 8 positions

          72 1 --- 010000

          R := R<8..15>_R<0..7>
```

.....

slc.8 R

```
   >>> shift left circular by 8 positions

          = src.8  R
```

.....

src.9 R

```
   >>> shift right circular by 9 positions

          72 1 --- 010001

          R := R<7..15>_R<0..6>
```

.....

slc.7 R

```
   >>> shift left circular by 7 positions

          = src.9  R
```

.....

src.10 R

```
   >>> shift right circular by 10 positions

          72 1 --- 010010

          R := R<6..15>_R<0..5>
```

.....

slc.6 R

```
   >>> shift left circular by 6 positions

          = src.10 R
```

.....

src.11 R

```
   >>> shift right circular by 11 positions

          72 1 --- 010011

          R := R<5..15>_R<0..4>
```

```
.....................................................................

    slc.5  R
       >>> shift left circular by 5 positions

           = src.11 R

.....................................................................

    src.12 R
       >>> shift right circular by 12 positions

           72 1 --- 010100

           R := R<4..15>_R<0..3>

.....................................................................

    slc.4  R
       >>> shift left circular by 4 positions

           = src.12 R

.....................................................................

    src.13 R
       >>> shift right circular by 13 positions

           72 1 --- 010101

           R := R<3..15>_R<0..2>

.....................................................................

    slc.3  R
       >>> shift left circular by 3 positions

           = src.13 R

.....................................................................

    src.14 R
       >>> shift right circular by 14 positions

           72 1 --- 010110

           R := R<2..15>_R<0..1>

.....................................................................

    slc.2  R
       >>> shift left circular by 2 positions

           = src.14 R

.....................................................................
```

src.15 R
>>> shift right circular by 15 positions

72 1 --- 010111

R := R<1..15>_R<0>

.....

slc.1 R
>>> shift left circular by 1 position

= src.15 R

.....

src.16 R
>>> shift right circular by 16 positions

72 0 --- 010000

R := R

.....

slc.0 R
>>> shift left circular by 0 positions

= src.16 R

.....

saz R
>>> shift left arithmetically with zero

72 1 --- 000100

H_R := R_0
flags: V

The register is shifted one position to the left. The rightmost bit is cleared. The leftmost bit of the former contents of the register goes into the H indicator. If the values of the leftmost register bit and the H indicator after the execution of the shift differ, the V indicator is set to 1.

.....

sah R
>>> shift left arithmetically with H

72 1 --- 000101

H_R := R_H
flags: V

The register is shifted one position to the left. The rightmost bit gets the former value of the H indicator. The leftmost bit of the former register value goes into the H indicator. If the values of the leftmost

bit of the register and the H indicator after the execution of the shift differ, the V indicator is set to 1.

.....

sax R

>>> shift left arithmetically with X

```
72 1 --- 000110

H_R := R_X
flags: V
```

The register is shifted one position to the left. The rightmost bit gets the value of the X indicator. The H indicator is set according to the former value of the leftmost bit. If the values of the leftmost register bit and the H indicator after the execution of the shift differ, the V indicator is set to 1.

.....

srz R

>>> shift right with zero

```
72 1 --- 001000

R_H := 0_R
```

The register is shifted one position to the right. The leftmost bit of the register is cleared. The rightmost bit of the former value of the register goes into H indicator.

.....

srh R

>>> shift right with H

```
72 0 --- 000111

R_H := H_R
```

The register is shifted one position to the right. The leftmost bit of the register gets the former value of H. The rightmost bit of the register is shifted into H.

.....

srx R

>>> shift right with X

```
72 1 --- 000111

R_H := X_R
```

The register is shifted one position to the right. The value of the leftmost bit is set to the former X value. The rigtmost bit of the register goes to the H indicator.

.....

slz R

>>> shift left with zero

72 0 --- 000100

H_R := R_0

The register is shifted one position to the left. The new value of the rightmost bit is zero. The H indicator is set according to the former contents of the leftmost bit.

.....

slh R

>>> shift left with H

72 0 --- 000101

H_R := R_H

The register is shifted one position to the left. The former value of H goes into the rightmost bit and the new H value is set to the former contents of the leftmost bit.

.....

slx R

>>> shift left with X

72 0 --- 000110

H_R := R_X

The register is shifted one position to the left. The rightmost bit of the register is set according to the value of X, and the contents of the leftmost bit is copied to the H indicator.

.....

vii.7 Comparing instructions

All comparing instruction affect three bits of r0: L, E and G. Exactly one of these bits is set to one, two other are cleared. If the value of the first argument is graeter than the value of the second argument G is set to one. If case of the equality E is set, and when the first argument's value is less than the second argument's value L is set.

cmpn R,SI
>>> compare numeric values

```
43 - --- ------

R ? Q
flags: L,E,G
```

The instruction compares the short integer contained in the register with the second argument's value treated as a short integer and sets the flags.

.....

cmps R,SN
>>> compare with short numeric

```
64 - --- ------

R ? Q
flags: L,E,G
```

The instruction compares the short integer contained in the register with the second argument's value, and sets the proper flags.

.....

cmpa R,SI
>>> compare addresses

```
54 - --- ------

R ? Q
flags: L,E,G
```

The instruction compares a bit string contained in the register with the second argument's value and sets the proper flags. The instruction can be used for address comparing as well.

.....

vii.8 Byte instructions

The group of byte instructions slightly violates the rules of argument determination. The second argument here is not a word (or words) but a single byte. The byte-addresses, represented by 16-bit values, can be used only to label bytes in words 0 - 32K-1. The fifteen older bits of byte-address are used as word-address, and the last bit determines the byte in the word. The left byte has the index zero, and the right one the index one. Clearly, if the second argument has a single indirect form its primary value is a byte-address. If a double indirect argument is used its primary value is a word-address which points to the word containing the byte-address.

Note that labels defined in the byte sections can stand for byte-addresses. If other arguments are used by the programmer to acces a single byte, the proper byte-addresses must be computed in the run-time. See also section viii of part one.

In the instruction semantic description Q'' stands for the byte defined by the second argument.

Example

Assume that we have the word-address of a structure placed in register r2. If we want to fetch the seventh byte of the structure, we must convert the structure address into byte-address form. The following instructions can be used:

```
setam     r2
l.byte r1,(r2+7)
```

The address conversion takes place dynamically; the byte-address is never stored in any register here.

l.byte R,DI

>>> load byte

```
55 - --- ------

R<8..15> := Q''
```

The byte described by the second argument is placed in bits 8..15 of the specified register. The left byte of the register is not changed. The address space of SI is used for fetching the byte.

.....

s.byte R,DI

>>> store byte

```
56 - --- ------

Q'' := R<8..15>
```

The right byte of the register is stored in the second argument. The indicated address space is used.

.....

c.byte R,DI
>>> compare bytes

57 - --- ------

R<8..15> ? Q''
flags: L,E,G

The right byte of the register is compared with the byte described by the second argument. The bytes are treated as non-negative numbers. The indicated address space is used for the evaluation of the second argument.

.....

vii.9 Conditional instructions

A conditional instruction usually performs a simple action, sometimes empty, and tests a condition, setting the skip indicator if the condition is not satisfied. The instruction mnemonics are designed in the form: if.---, where the dashes stand for a shortened name of the test performed. See also the descriptions of the processor and of the machine instruction execution for details.

if.se. R,DI

```
   >>> if section successfully entered

       27 - --- ------

       if  Q'.and.R = R
          then S := 1
          else Q' := Q' .ior. R
```

The bit-by-bit product of the register and the second argument's value is compared against the original register contents. The next instruction is then skipped (equality), or the logical 'or' of the contents of the register and of the second argument's value is stored in the second argument. All these actions are performed within a single bus reservation cycle: the instruction performs an indivisible action on memory and can be used as Test&Branch or Test&Set statement. The second argument is located in the indicated address space.

.....

if.nl. R,DI

```
   >>> if section not locked

       31 - --- ------

       if  Q'.and.R = R
          then S := 1
```

The condition tested is the same as in the case of the previous instruction. The second argument is unchanged, only the skip indicator can be set here. The indicated address space is used.

.....

if.anc R,SI

```
   >>> if any (of indicated bits) cleared

       30 - --- ------

       if  R.and.Q = Q
          then S := 1
```

The bit-by-bit logical product of the first and the second argument value is compared with the second argument's value. In case of equality the S indicator is set.

```
.....................................................................

    if.nz
       >>> if not zero

           = if.anc r0,X'8000'

.....................................................................

    if.pl
       >>> if plus

           = if.anc r0,X'4000'

.....................................................................

    if.ge
       >>> if greater or equal

           = if.anc r0,X'0800'

.....................................................................

    if.ne
       >>> if not equal

           = if.anc r0,X'0400'

.....................................................................

    if.le
       >>> if less or equal

           = if.anc r0,X'0200'

.....................................................................

    if.anz   DI
       >>> if advanced not zero

           77 - 111 ------

           Q := Q + 1
           if Q=0
              then S := 1
```

The memory word described by the second argument is incremented by one. The next instruction will be executed if the result of incrementation is not zero. Note that arithmetic flags are not changed.

```
.....................................................................

    if.inz R,SN
       >>> if incremented not zero

           61 - --- ------

           R := R + Q
           if R=0
```

```
            then S := 1
```

The second argument's value (extended to the 16-bit value) is added to the register. The next instruction will be executed if the result is non-zero. No flag of r0 is altered.

.....

if.dnz R,SN

```
   >>> if decremented not zero

       = if.inz R,-SN
```

.....

if.nzr R

```
   >>> if not zero in register

       = if.inz R,0
```

.....

if.als R,SI

```
   >>> if all (of indicated bits) set

       33 - --- ------

       if  R.and.Q != Q
          then S := 1
```

The bit-by-bit product of the register and second argument's value is compared with the second argument's value. In case of equality the next instruction will be executed.

.....

if.ans R,SI

```
   >>> if any (of indicated bits) set

       34 - --- ------

       if  R.and.Q = 0
          then S := 1
```

If the bit-by-bit logical product of the first and the second argument's value is zero the S indicator is set. Flags of r0 are not changed.

.....

if.nsm R,SI

```
   >>> if not same under mask

       32 - --- ------

       if  R.and.r7 = Q.and.r7
          then S := 1
```

Bits of the register and bits of the second argument's value that correspond to the set bits of the r7 register are compared. In case of equality the skip indicator is set.

.....

if.afs BA
>>> if indicated bits in arithmetic flags set

```
71 0 0-- ------

if r0<0..7>.and.Q != Q
   then S := 1
```

The bit-by-bit logical product of the AF register and the second argument's value is compared with the second argument's value. In case of equality the next instruction will be executed.

.....

if.zr
>>> if zero

```
= if.afs X'80'
```

.....

if.ng
>>> if negative

```
= if.afs X'40'
```

.....

if.vs
>>> if V set

```
= if.afs X'20'
```

.....

if.cs
>>> if C set

```
= if.afs X'10'
```

.....

if.lt
>>> if less than

```
= if.afs X'08'
```

.....

if.eq
>>> if equal

```
= if.afs X'04'
```

.....

if.gt
>>> if greater than

```
= if.afs X'02'
```

.....

if.hs
>>> if H set

```
= if.afs X'01'
```

.....

if.sbs BA
>>> if indicated bits in scratch byte set

```
71 1 0-- ------

if  r0<8..15>.and.Q != Q
   then S := 1
```

The bit-by-bit logical product of the SB register and the second argument's value is compared with the second argument's value. In case of equality the next instruction will be executed.

.....

if.xs
>>> if X set

```
= if.sbs X'80'
```

.....

vii.10 Branching instructions

Note that assembled values of the SR arguments are adjusted by the value of the location counter to provide the correct interpretation.

i.bnz R,SR
>>> increment & branch on not zero

62 - --- ------

```
R := R + 1
if R != 0
   then IC := IC + Q primary value
```

The register is incremented. If the result is non-zero the instruction counter is incremented by the second argument's primary value.

.....

d.bnz R,SR
>>> decrement & branch on not zero

63 - --- ------

```
R := R - 1
if R != 0
   then IC := IC + Q primary value
```

The register is decremented by one. If the result is non-zero the instruction counter is incremented by the second argument's primary value.

.....

b.uc SR
>>> branch unconditionally

70 - 000 ------

```
IC := IC + Q primary value
```

The instruction counter is incremented by the second argument's primary value.

.....

buc DI
>>> branch unconditionally

74 - 000 ------

```
IC := address of Q
```

The address of the second argument is moved to the instruction counter.

.....

bzr **DI**

>>> branch on zero

```
74 - 100 ------

if Z=1
   then IC := address of Q
```

If the Z bit of r0 is set the address of the second argument is moved to the instruction counter.

f the second argument is moved
to the instruction counter.

.....

bng **DI**

>>> branch on negative

```
74 - 101 ------

if N=1
   then IC := address of Q
```

If the N bit of r0 is set the address of the second argument is moved to the instruction counter.

.....

b.vs **SR**

>>> branch on V set

```
70 - 100 ------

if V=1
   then IC := IC + Q primary value
V := 0
```

If the V bit of r0 is set the second argument's primary value is added to the instruction counter. The V bit is cleared.

.....

b.cs **SR**

>>> branch on C set

```
70 - 111 ------

if C=1
   then IC := IC + Q primary value
```

If the C bit of r0 is set the second argument's primary value is added to the instruction counter.

.....

b.lt **SR**
>>> branch on less than

70 - 001 ------

if L=1
then IC := IC + Q primary value

If the L bit of r0 is set the second argument's primary value is added to the instruction counter.

.....

blt **DI**
>>> branch on less than

74 - 001 ------

if L=1
then IC := address of Q

If the L bit of r0 is set the address of the second argument is moved to the instruction counter.

.....

b.eq **SR**
>>> branch on equal

70 - 010 ------

if E=1
then IC := IC + Q primary value

If the E bit of r0 is set the second argument's primary value is added to the instruction counter.

.....

beq **DI**
>>> branch on equal

74 - 010 ------

if E=1
then IC := address of Q

If the E bit of r0 is set the address of the second argument is moved to the instruction counter.

.....

bne **DI**
>>> branch on not equal

74 - 110 ------

if E=0
then IC := address of Q

If the E bit of r0 is cleared the address of the second argument is moved to the instruction counter.

.....

b.gt **SR**

>>> branch on greater than

```
70 - 011 ------

if G=1
   then IC := IC + Q primary value
```

If the G bit of r0 is set the second argument's primary value is added to the instruction counter.

.....

bgt **DI**

>>> branch on greater than

```
74 - 011 ------

if G=1
   then IC := address of Q
```

If the G bit of r0 is set the address of the second argument is moved to the instruction counter.

.....

b.hs **SR**

>>> branch on H set

```
70 - 110 ------

if H=1
   then IC := IC + Q primary value
```

If the H bit of r0 is set the second argument's primary value is added to the instruction counter.

.....

b.xs **SR**

>>> branch on X set

```
70 - 101 ------

if X=1
   then IC := IC + Q primary value
```

If the X bit of r0 is set the second argument's primary value is added to the instruction counter.

.....

callm DI
>>> call a subprogram with trace in memory

```
74 - 111 ------

Q := IC
IC := address of Q(1)
```

The instruction counter is stored in the second argument. Then the second argument address is incremented by one and is moved to the instruction counter.

.....

call R,DI
>>> call a subprogram

```
26 - --- ------

R := IC
IC := address of Q
```

The instruction counter is copied to the register. Then the address of the second argument is moved to the instruction counter.

.....

vii.11 Floating-point arithmetic

A floating point value consists of a 40-bit coefficient and of an 8-bit exponent - it is placed in 3 consecutive words. The coefficient is a two-complement fractional number and the exponent is expressed in the form of a two-complementary 8-bit integer.

A correct floating point number is always normalized, i.e. the first two bits of any non-zero number are different. The representation of zero consists of 48 binary zeroes.

The range for absolute values of floating-point numbers is:

1.47E-39 .. 1.70E38

and the relative accuracy of arithmetic operations is:

3.73E-9

All the arithmetic operations except division produce rounded results. The bit used for rounding (the 41st bit of the resusults. The bit used for rounding (the 41st bit of the result) is placed then in C indicator.

The exceptions of underflow or overflow, zero divisor and unnormalized argument are detected and cause interrupts.

norm

>>> normalize floating

71 1 1-- ------

r1_r2_r3 := r1_r2_r3
flags: Z,N,C

The bits of r1,r2 and the left byte of r3 (the coefficient) are shifted left until the first two bits of r1 are different. The number of positions shifted is added to the right byte of r3 (the exponent). If the coefficient was zero the exponent is also cleared. The C indicator is cleared.

.....

addf DI

>>> add floating

37 - 100 ------

r1_r2_r3 := r1_r2_r3 + Q(0)_Q(1)_Q(2)
flags: Z,N,C

The floating-point number described by the second argument is added to the floating point number placed in r1, r2 and r3. Flags are set accordingly.

.....

subf DI
>>> subtract floating

```
37 - 101 ------

r1_r2_r3 := r1_r2_r3 - Q(0)_Q(1)_Q(2)
flags: Z,N,C
```

The floating-point number defined by the second argument is subtracted from the number in registers r1,r2 and r3.

.....

mulf DI
>>> multiply floating

```
37 - 110 ------

r1_r2_r3 := r1_r2_r3 * Q(0)_Q(1)_Q(2)
flags: Z,N,C
```

The number in registers r1, r2 and r3 is multiplied by the floating-point number defined by the second argument.

.....

divf DI
>>> divide floating

```
37 - 111 ------

r1_r2_r3 := r1_r2_r3 / Q(0)_Q(1)_Q(2)
flags: Z,N,C
```

The number placed in r1, r2 and r3 is divided into the floating-point number defined by the second argument. The C indicator is cleared.

.....

Appendix A

The Mera-400 character set

	0	1	2	3	4	5	6	7
0	nul	dle		0	@	P		p
1	soh	dc1	!	1	A	Q	a	q
2	stx	dc2	"	2	B	R	b	r
3	etx	dc3	#	3	C	S	c	s
4	eot	dc3	$	4	D	T	d	t
5	enq	nak	%	5	E	U	e	u
6	ack	syn	&	6	F	V	f	v
7	bel	etb	'	7	G	W	g	w
8	bs	can	(	8	H	X	h	x
9	ht	em	)	9	I	Y	i	y
A	lf	sub	*	:	J	Z	j	z
B	vt	esc	+	;	K	[	k	{
C	ff	fs	,	<	L	\	l	|
D	cr	gs	-	=	M	]	m	}
E	so	rs	.	>	N	^	n	~
F	si	us	/	?	O	_	o	del

Appendix B

The shortened instruction list

```
set    R,SI  20 - --- ------      set register
load.  R,DI  21 - --- ------      load register
mld    R,SI  22 - --- ------      mask & load
push   R,SI  23 - --- ------      push value onto stack
store  R,DI  24 - --- ------      store register
store. R,DI  25 - --- ------      store register
call   R,DI  26 - --- ------      call a subprogram
if.se. R,DI  27 - --- ------      if section successfully entered
if.anc R,SI  30 - --- ------      if any (of indicated bits) cleared
if.nz        30 0 000 000000 8000 if not zero
if.pl        30 0 000 000000 4000 if plus
if.ge        30 0 000 000000 0800 if greater or equal
if.ne        30 0 000 000000 0400 if not equal
if.le        30 0 000 000000 0200 if less or equal
if.nl. R,DI  31 - --- ------      if section not locked
if.nsm R,SI  32 - --- ------      if not same under mask
if.als R,SI  33 - --- ------      if all (of indicated bits) set
if.ans R,SI  34 - --- ------      if any (of indicated bits) set
o.rq   R,SI  35 - --- ------      output request
i.rq   R,SI  36 - --- ------      input request
addl     DI  37 - 000 ------      add long
subl     DI  37 - 001 ------      subtract long
mul      DI  37 - 010 ------      multiply
div      DI  37 - 011 ------      divide
addf     DI  37 - 100 ------      add floating
subf     DI  37 - 101 ------      subtract floating
mulf     DI  37 - 110 ------      multiply floating
divf     DI  37 - 111 ------      divide floating
add    R,SI  40 - --- ------      add
adc    R,SI  41 - --- ------      add with carry
sub    R,SI  42 - --- ------      subtract
cmpn   R,SI  43 - --- ------      compare numeric values
ior    R,SI  44 - --- ------      logical inclusive or
iorm.  R,DI  45 - --- ------      logical inclusive or in memory
and    R,SI  46 - --- ------      logical and
andm.  R,DI  47 - --- ------      logical and in memory
erb    R,SI  50 - --- ------      erase bits
erbm.  R,DI  51 - --- ------      erase bits in memory
xor    R,SI  52 - --- ------      logical exclusive or
xorm.  R,DI  53 - --- ------      logical exclusive or in memory
cmpa   R,SI  54 - --- ------      compare addresses
l.byte R,DI  55 - --- ------      load byte
s.byte R,DI  56 - --- ------      store byte
c.byte R,DI  57 - --- ------      compare bytes
adds   R,SN  60 - --- ------      add short
subs   R,SN  60 - --- ------      subtract short
inc    R     60 0 --- 000001      increment
dec    R     60 1 --- 000001      decrement
if.inz R,SN  61 - --- ------      if incremented not zero
if.dnz R,SN  61 - --- ------      if decremented not zero
if.nzr R     61 0 --- 000000      if not zero in register
i.bnz  R,SR  62 - --- ------      increment & branch on not zero
d.bnz  R,SR  63 - --- ------      decrement & branch on not zero
cmps   R,SN  64 - --- ------      compare with short
lds    R,SN  65 - --- ------      load short
```

clr	R	65	0	---	000000	clear register
l.rg	R,SR	66	-	---	------	load register
s.rg	R,SR	67	-	---	------	store register
b.uc	SR	70	-	000	------	branch unconditionally
b.lt	SR	70	-	001	------	branch on less than
b.eq	SR	70	-	010	------	branch on equal
b.gt	SR	70	-	011	------	branch on greater than
b.vs	SR	70	-	100	------	branch on V set
clv		70	0	100	000000	clear V
b.xs	SR	70	-	101	------	branch on X set
b.hs	SR	70	-	110	------	branch on H set
b.cs	SR	70	-	111	------	branch on C set
if.afs	BA	71	0	0--	------	if indicated bits in AF set
if.zr		71	0	010	000000	if zero
if.ng		71	0	001	000000	if negative
if.vs		71	0	000	100000	if V set
if.cs		71	0	000	010000	if C set
if.lt		71	0	000	001000	if less than
if.eq		71	0	000	000100	if equal
if.gt		71	0	000	000010	if greater than
if.hs		71	0	000	000001	if H set
ptr	BA	71	0	1--	------	program trap
if.sbs	BA	71	1	0--	------	if indicated bits in scratch byte set
if.xs		71	1	010	000000	if X set
norm		71	1	1--	------	normalise floating
micr	R	72	0	---	000000	move instruction counter to register
cllb	R	72	0	---	000001	clear left byte
msx	R	72	0	---	000010	move sign bit to X
ngn	R	72	0	---	000011	negate numeric value
slz	R	72	0	---	000100	shift left with zero
slh	R	72	0	---	000101	shift left with H
slx	R	72	0	---	000110	shift left with X
srh	R	72	0	---	000111	shift right with H
rvb	R	72	0	---	001000	reverse bits
mpfr	R	72	0	---	001001	move program flags to the register
src.16	R	72	0	---	010000	shift right circular by 16 positions
slc.0	R	72	0	---	010000	shift left circular by 0 positions
src.1	R	72	0	---	010001	shift right circular by 1 position
slc.15	R	72	0	---	010001	shift left circular by 15 positions
src.2	R	72	0	---	010010	shift right circular by 2 positions
slc.14	R	72	0	---	010010	shift left circular by 14 positions
src.3	R	72	0	---	010011	shift right circular by 3 positions
slc.13	R	72	0	---	010011	shift left circular by 13 positions
src.4	R	72	0	---	010100	shift right circular by 4 positions
slc.12	R	72	0	---	010100	shift left circular by 12 positions
src.5	R	72	0	---	010101	shift right circular by 5 positions
slc.11	R	72	0	---	010101	shift left circular by 11 positions
src.6	R	72	0	---	010110	shift right circular by 6 positions
slc.10	R	72	0	---	010110	shift left circular by 10 positions
src.7	R	72	0	---	010111	shift right circular by 7 positions
slc.9	R	72	0	---	010111	shift left circular by 9 positions
mprr	R	72	1	---	000000	move panel register to the register
clrb	R	72	1	---	000001	clear right byte
mpx	R	72	1	---	000010	move parity bit to X
ngc	R	72	1	---	000011	negate with carry
saz	R	72	1	---	000100	shift left arithmetically with zero
sah	R	72	1	---	000101	shift left arithmetically with H
sax	R	72	1	---	000110	shift left arithmetically with X
srx	R	72	1	---	000111	shift right with X

```
srz    R     72 1 --- 001000      shift right with zero
mrpf   R     72 1 --- 001001      move the register to the program flags
nop          72 1 000 001001      no operation
src.8  R     72 1 --- 010000      shift right circular by 8 positions
slc.8  R     72 1 --- 010000      shift left circular by 8 positions
swab   R     72 1 --- 010000      swap bytes
src.9  R     72 1 --- 010001      shift right circular by 9 positions
slc.7  R     72 1 --- 010001      shift left circular by 7 positions
src.10 R     72 1 --- 010010      shift right circular by 10 positions
slc.6  R     72 1 --- 010010      shift left circular by 6 positions
src.11 R     72 1 --- 010011      shift right circular by 11 positions
slc.5  R     72 1 --- 010011      shift left circular by 5 positions
src.12 R     72 1 --- 010100      shift right circular by 12 positions
slc.4  R     72 1 --- 010100      shift left circular by 4 positions
src.13 R     72 1 --- 010101      shift right circular by 13 positions
slc.3  R     72 1 --- 010101      shift left circular by 3 positions
src.14 R     72 1 --- 010110      shift right circular by 14 positions
slc.2  R     72 1 --- 010110      shift left circular by 2 positions
src.15 R     72 1 --- 010111      shift right circular by 15 positions
slc.1  R     72 1 --- 010111      shift left circular by 1 position
halt         73 - 000 ------      stop the computer
reset        73 - 001 ------      reset the configuration
cbpi         73 - 010 ----00      clear both program interrupts
rlpi         73 - 010 ----01      request low program interrupt
rhpi         73 - 010 ----10      request high program interrupt
rbpi         73 - 010 ----11      request both program interrupts
if.hsi       73 0 011 ------      if high interrupt sent and ignored
if.lsi       73 1 011 ------      if low interrupt sent and ignored
exit         73 - 100 ------      exit interrupt routine
buc      DI  74 - 000 ------      branch unconditionally
blt      DI  74 - 001 ------      branch on less than
beq      DI  74 - 010 ------      branch on equal
bgt      DI  74 - 011 ------      branch on greater than
bzr      DI  74 - 100 ------      branch on zero
bng      DI  74 - 101 ------      branch on negative
bne      DI  74 - 110 ------      branch on not equal
callm    DI  74 - 111 ------      call a subprogram with trace in memory
ldl      DI  75 - 000 ------      load long
ldf      DI  75 - 001 ------      load floating
lda      DI  75 - 010 ------      load all
ldh      DI  75 - 011 ------      load high
ldl.     DI  75 - 100 ------      load long
ldf.     DI  75 - 101 ------      load floating
lda.     DI  75 - 110 ------      load all
ldh.     DI  75 - 111 ------      load high
strl     DI  76 - 000 ------      store long
strf     DI  76 - 001 ------      store floating
stra     DI  76 - 010 ------      store all
strh     DI  76 - 011 ------      store high
strl.    DI  76 - 100 ------      store long
strf.    DI  76 - 101 ------      store floating
stra.    DI  76 - 110 ------      store all
strh.    DI  76 - 111 ------      store high
ldsi     DI  77 - 000 ------      load space index
ldpm     DI  77 - 001 ------      load processor mask
stpi     DI  77 - 010 ------      store processor interrupts
ldpi     DI  77 - 011 ------      load processor interrupts
swch.    DI  77 - 100 ------      switch to another program
setam    SI  77 - 101 ------      set argument modifier
```

```
clrm     DI  77 - 110 ------      clear memory word
incm     DI  77 - 111 ------      increment memory word
if.anz   DI  77 - 111 ------      if advanced not zero
```

Appendix C

The alphabetical index of the Gass instructions

adc R,SI 37
add R,SI 34
addf DI 62
addl DI 35
adds R,SN 35
and R,SI 38
andm. R,DI 38

b.cs SR 58
b.eq SR 59
b.gt SR 60
b.hs SR 60
b.lt SR 59
b.uc SR 57
b.vs SR 58
b.xs SR 61
beq DI 59
bgt DI 60
blt DI 59
bne DI 60
bng DI 58
buc DI 57
bzr DI 58

c.byte R,DI 51
call R,DI 61
callm DI 61
cbpi 21
cllb R 23
clr R 24
clrb R 24
clrm DI 24
clv 23
cmpa R,SI 49
cmpn R,SI 49
cmps R,SN 49

d.bnz R,SR 57
dec R 35
div DI 36
divf DI 63

erb R,SI 39
erbm. R,DI 40
exit 19

halt 19

i.bnz R,SR 57
i.rq R,SI 19
if.afs BA 55
if.als R,SI 54
if.anc R,SI 52
if.ans R,SI 54
if.anz DI 53

if.cs 55
if.dnz R,SN 54
if.eq 56
if.ge 53
if.gt 56
if.hs 56
if.hsi 22
if.inz R,SN 54
if.le 53
if.lsi 22
if.lt 56
if.ne 53
if.ng 55
if.nl. R,DI 52
if.nsm R,SI 55
if.nz 53
if.nzr R 54
if.pl 53
if.sbs BA 56
if.se. R,DI 52
if.vs 55
if.xs 56
if.zr 55
inc R 35
incm DI 24
ior R,SI 38
iorm. R,DI 39

l.byte R,DI 51
l.rg R,SR 28
lda. DI 31
lda DI 31
ldf. DI 31
ldf DI 30
ldh. DI 32
ldh DI 32
ldl. DI 30
ldl DI 29
ldpi DI 21
ldpm DI 21
lds R,SN 28
ldsi DI 21
load. R,DI 29

micr R 26
mld R,SI 24
mpfr R 26
mprr R 26
mpx R 25
mrpf R 25
msx R 25
mul DI 36
mulf DI 63

ngc R 37
ngn R 34
nop 26
norm 62

o.rq R,SI 17

ptr BA 23
push R,SI 25

rbpi 20
reset 22
rhpi 20
rlpi 20
rvb R 38

s.byte R,DI 50
s.rg R,SR 28
sah R 46
sax R 46
saz R 45
set R,SI 28
setam SI 26
slc.0 R 45
slc.10 R 42
slc.11 R 42
slc.12 R 42
slc.13 R 41
slc.14 R 41
slc.15 R 41
slc.1 R 45
slc.2 R 45
slc.3 R 44
slc.4 R 44
slc.5 R 44
slc.6 R 43
slc.7 R 43
slc.8 R 43
slc.9 R 43
slh R 47
slx R 48
slz R 47
src.10 R 43
src.11 R 44
src.12 R 44
src.13 R 44
src.14 R 45
src.15 R 45
src.16 R 45
src.1 R 41
src.2 R 41
src.3 R 41
src.4 R 42
src.5 R 42
src.6 R 42
src.7 R 42
src.8 R 43
src.9 R 43
srh R 47
srx R 47
srz R 46
store. R,DI 29
store R,DI 30
stpi DI 21

stra.	DI	32
stra	DI	31
strf.	DI	31
strf	DI	30
strh.	DI	32
strh	DI	32
strl.	DI	30
strl	DI	29
sub	R,SI	35
subf	DI	63
subl	DI	36
subs	R,SN	35
swab	R	24
swch.	DI	20
xor	R,SI	39
xorm.	R,DI	39